Superword-Level Parallelism in the Presence of Control Flow

Jaewook Shin, Mary Hall and Jacqueline Chame

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{jaewook,mhall,jchame}@isi.edu

Abstract

In this paper, we describe how to extend the concept of superword-level parallelization (SLP), used for multimedia extension architectures, so that it can be applied in the presence of control flow constructs. Superword-level parallelization involves identifying scalar instructions in a large basic block that perform the same operation, and, if dependences do not prevent it, combining them into a superword operation on a multi-word object. A key insight is that we can use techniques related to optimizations for architectures supporting predicated execution, even for multimedia ISAs that do not provide hardware predication. We derive large basic blocks with predicated instructions to which SLP can be applied. We describe how to minimize overheads for superword predicates and re-introduce control flow for scalar operations. We discuss other extensions to SLP to address common features of real multimedia codes. We present automatically-generated performance results on 8 multimedia codes to demonstrate the power of this approach. We observe speedups ranging from 1.97X to 15.07X as compared to both sequential execution and SLP alone.

1 Introduction

Many modern microprocessors incorporate an expanded instruction set specifically targeting the requirements of multimedia applications, with a functional unit that can operate on aggregate objects to perform bit-level operations, or SIMD parallel operations on variable-sized fields in the object (*e.g.,* 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than the size of a machine word, then they are called *superwords* [16].

Initially, the conventional wisdom was that the appropriate compiler technology for multimedia extensions would borrow heavily from automatic vectorization [25, 5, 7]. More recently, Larsen and Amarasinghe demonstrated that unique features of the architectures and the target multimedia applications suggest that a different strategy, called *superword-level parallelization* (SLP), can often yield more effective optimization [16]. Architecturally, multimedia extensions support *simultaneous* SIMD execution on *short "vectors"* (128 bits on a PowerPC AltiVec), rather than *pipelined* SIMD execution of *long vectors* on vector architectures. As a consequence, in a multimedia extension architecture, initiating a parallel computation has fairly low overhead, and can be mixed with sequential instructions in an efficient manner. On the other hand, the benefits over sequential execution are not as great on a short "vector", and whatever overheads exist must be carefully managed.

As compared to vectorization, rather than relying on high-level loop transformations, SLP involves packing *isomorphic* instructions and their associated data into superwords, possibly performing loop unrolling to expose parallelism. While this approach is simple and effective, it only identifies parallelism within a basic block. The following simple and inherently parallel loop would not be parallelized:

```
for (i=0; i<16; i++)
    if (a[i] != 0)
        b[i]++;
```

Superword-level parallelization in the presence of control flow is still an open issue: how to identify parallelism in the presence of control flow, how to best use multimedia ISA features, and how to avoid overheads that lead to performance degradations as compared to scalar code. Support for parallelizing control flow is important to multimedia applications. As one data point, control flow appears in key computations in 6 of the 11 codes in the UCLA MediaBench [18], comprising on average over 40% of their execution time.

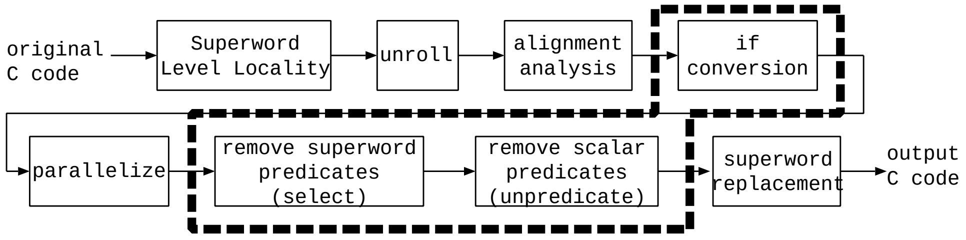


Figure 1. An SLP-based compiler that supports control flow.

We are developing a compiler extending MIT's SLP compiler for both the PowerPC AltiVec and a research architecture called DIVA that uses processing-in-memory technology to exploit the high memory bandwidth when processing logic is combined with large amounts of memory [12, 8]. In this paper, we examine the impact of incorporating parallelization of control flow for multimedia architectures. We describe our compiler implementation that attains significant performance improvements over SLP and sequential execution in the presence of control flow, ranging from 1.97X to 15.07X for 8 multimedia codes. A key insight in this work is that we can borrow heavily from optimizations developed for architectures supporting wide-issue instruction-level parallelism and predicated execution, such as, for example, the Itanium family of processors [14], *even for architectures such as the AltiVec that do not support predication*. There are two reasons why similar optimization techniques can be used for these two distinct classes of architectures:

- SLP and ILP optimizations operate within basic blocks. Control flow limits the size of basic blocks, and thus limits optimization opportunities. We derive large basic blocks with predicated instructions to which SLP can be applied.
- A commonality in multimedia extension ISAs is what we will call a `select` operation for merging the results of different control flow paths. Based on the value of a boolean superword, individual fields from two different inputs are combined and committed to a final result. Thus, `select` instructions appear similar to predicated instructions, even though the underlying hardware mechanisms to implement the two are very different.

The remainder of the paper is organized as follows. Section 2 presents an overview of the compiler techniques required for SLP optimizations in the presence of control flow. Section 3 presents the two main new algorithms, based on related algorithms supporting ILP and predicated execution. Section 4 describes relevant parallelization and code generation issues. Section 5 presents results of the implementation on 8 multimedia codes, yielding speedups ranging from 1.97X to 15.07X over sequential execution. Section 6 describes related work, followed by a conclusion.

2 Motivation

In Figure 1, we present the collection of analyses and transformations to exploit SLP in the presence of control flow. Analyses from the standard SLP compiler are described in [16, 17]. We also perform the compiler-controlled caching of [23] in two distinct phases. Superword level locality analysis identifies the potential for superword register reuse and guides loop unrolling and unroll-and-jam. In a later phase, superword replacement exploits the exposed reuse by removing redundant memory accesses.

The analyses within the dashed box represent control-flow extensions beyond what is required for SLP restricted to a basic block. Figure 2 illustrates the goals of these analyses and transformations using the C code snippet from Chroma Key in Figure 2(a).

As shown in 2(b), the code is unrolled by a factor of four, based on the assumption that the superword register width is sixteen bytes and the array type sizes are four bytes. Next, *if-conversion* using Park and Schlansker's algorithm [22] is applied to convert control dependences into data dependences. Now, associated with each instruction is a *predicate*, shown in parenthesis at the end of the instruction, that captures the conditions that must be true for the instruction to execute. The `pset` instruction initializes the value of the predicates `pT` and `pF` based on the value of the condition represented by `comp`.

After if-conversion, the loop body becomes one basic block of predicated instructions. A modified version of the SLP parallelizer, which packs together isomorphic instructions with their predicates, derives a mix of predicated scalar and superword instructions. The resulting code is shown in Figure 2(c). In Figure 2(d), we show how a superword `select` operation can be used to select individual fields from two superword definitions according to the value of a superword predicate variable. Concretely, the effect of the `select` operation "`dst = select(src1, src2, mask)`", is to assign `src2` to `dst` for the fields where the corresponding `mask` bit is 1. Otherwise, `src1`

```
for(i=0; i<1024; i++){
    if(fore_blue[i] != 255){
        back_blue[i] = fore_blue[i];
        back_red[i+1] = back_red[i];
    }
}
```

(a) Original

```
for(i=0; i<1024; i+=4){
    comp = fore_blue[i] != 255;
    pT, pF = pset(comp);
    back_blue[i] = fore_blue[i];                (pT)
    back_red[i+1] = back_red[i];                (pT)
        . . .
}
```

(b) Unrolled and if-converted

```
for(i=0; i<1024; i+=4){
    v_comp = fore_blue[i:i+3] != (255,255,255,255);
    v_pT, v_pF = v_pset(v_comp);
    back_blue[i:i+3] = fore_blue[i:i+3];        (v_pT)
    pT1, pT2, pT3, pT4 = unpack(v_pT);
    back_red[i+1] = back_red[i];                (pT1)
    back_red[i+2] = back_red[i+1];              (pT2)
    back_red[i+3] = back_red[i+2];              (pT3)
    back_red[i+4] = back_red[i+3];              (pT4)
}
```

(c) Parallelized

```
for(i=0; i<1024; i+=4){
    v_comp = fore_blue[i:i+3] != (255,255,255,255);
    v_pT, v_pF = v_pset(v_comp);
    back_blue[i:i+3] = select(back_blue[i:i+3], fore_blue[i:i+3], v_pT)
    pT1, pT2, pT3, pT4 = unpack(v_pT);
    back_red[i+1] = back_red[i];                (pT1)
    back_red[i+2] = back_red[i+1];              (pT2)
    back_red[i+3] = back_red[i+2];              (pT3)
    back_red[i+4] = back_red[i+3];              (pT4)
}
```

(d) Select applied

```
for(i=0; i<1024; i+=4){
    v_comp = fore_blue[i:i+3] != (255,255,255,255);
    v_pT, v_pF = v_pset(v_comp);
    back_blue[i:i+3] = select(back_blue[i:i+3], fore_blue[i:i+3], v_pT)
    pT1, pT2, pT3, pT4 = unpack(v_pT);
    if(pT1) back_red[i+1] = back_red[i];
    if(pT2) back_red[i+2] = back_red[i+1];
    if(pT3) back_red[i+3] = back_red[i+2];
    if(pT4) back_red[i+4] = back_red[i+3];
}
```

(e) Unpredicated

Figure 2. Example illustrating steps of SLP compilation in the presence of control flow.

is assigned to `dst`. Figure 3 shows this graphically using superwords of 4 scalar elements. Note that the effect of this transformation is to execute both control flow paths and select the value from the one that would have executed in the scalar version of the code. Thus, the parallelization overhead includes the `select` instructions, and the cost of executing both paths. In Section 3.2, we describe how to minimize the number of `select` instructions to reduce this overhead.

$\boxed{3}\boxed{2}\boxed{3}\boxed{2}$ = SELECT($\boxed{2}\boxed{2}\boxed{2}\boxed{2}$, $\boxed{3}\boxed{3}\boxed{3}\boxed{3}$, $\boxed{1}\boxed{0}\boxed{1}\boxed{0}$);

Figure 3. Merging two superwords using a SELECT instruction

Next, we restore the control flow for the predicated scalar operations, as shown in Figure 2(e). While it is straightforward to insert control flow corresponding to the predicate on the instruction, this strategy could result in an enormous amount of additional branches as compared to the original scalar code. Thus, another important optimization is minimizing the branches, with an attempt to recover as close as possible the control flow of the original scalar code, as described in Section 3.3.

Discussion. The approach described above has been heavily influenced by features of the ISA of the target architectures, as well as the current organization of the SLP compiler, where we treat the SLP pass as a black box and feed it large basic blocks for parallelization. If the target architecture supported *masked* superword operations [24] and predicated scalar execution [22, 13], the code in Figure 2(c) would not need any further transformations for SLP. The DIVA ISA supports masked superword operations, but not predicated execution, and the PowerPC AltiVec, the other platform for our work, supports neither. Thus the compiler must eliminate the predicates on scalar instructions by restoring control flow, and for architectures including the AltiVec, replace the predicated superword instructions with `select` instructions that achieve the same effect.

If the architecture combined SLP support and predication, we could adapt recently-developed algorithms by Chuang et. al. to generate *phi-instructions* from the CFG of a scalar code to resolve the *multiple-definition problem* in architectures that support predicated execution [6]. Their phi-instruction is a scalar analog to the superword *select* instruction. While it is possible to use the phi-predicated code as an input to SLP, some scalar predicated operations would remain and scalar control flow would nevertheless need to be restored in architectures such as the AltiVec.

```
if (b[i]<0){
  a[i] = 1;
}else{
  a[i] = 0;
}
.. = a[i];
```

(a) scalar

```
Vp, Vnp = Vb < V0
Va = V1       (Vp)
Va = V0       (Vnp)
... = Va
```

(b) parallelized intermediate form

```
Vp, Vnp = Vb < V0
Va1 = V1
Va = select(Va, Va1, Vp)
Va2 = V0
Va = select(Va, Va2, Vnp)
... = Va
```

(c) naive generation of **select**

Figure 4. Merging two superword definitions

3 Algorithm

In this section, we present the two analyses and code transformation techniques outlined above. Following if-conversion, the resulting code contains large basic blocks of instructions, and in some cases, the instructions are predicated. The compiler's job is to remove these predicates. We discuss how superword predicates are removed by inserting `select` operations in Section 3.2. We describe how scalar predicates are removed through an algorithm we call *unpredicate* in Section 3.3. Prior to these descriptions, we introduce a few definitions in Section 3.1.

3.1 Definitions

Since SLP relies on if-conversion, there is a lot of similarity between the techniques described in this paper, and those used to compile for architectures supporting predicated execution. Thus, we borrow several concepts from Mahlke's work, defined as follows [20].

Definition 1 *A predicate hierarchy graph(PHG) is a directed acyclic graph representing nesting relations among predicates in a predicated basic block (after if-conversion has been applied).*

A PHG consists of two types of nodes, *predicate nodes* and *condition nodes*, and is constructed as follows. Starting with a single predicate node, each instruction is examined in textual order. For each instruction that defines predicates, such as, for example `pT, pF = pset(comp) (pParent);` above, at most two condition nodes are created, representing the true and false values of a comparison. For this example, condition nodes `comp` and $\neg$`comp` would be created. Edges are inserted from the predicate node for the predicate guarding the instruction to the condition nodes just created; for the example, edges from predicate node `pParent` to condition nodes `comp` and $\neg$`comp` are added. Predicate nodes representing `pT` and `pF` are also created, if they do not already exist. They may have been introduced into the PHG by a prior definition, in cases where multiple control flow paths merge. Then, edges are inserted from the condition nodes to the corresponding predicate nodes; in the example, there would be an edge inserted from condition node `comp` to predicate node `pT`, and from $\neg$`comp` to `pF`. This process is repeated for each instruction that defines a predicate. The resulting PHG permits analysis to reason about the relationship among predicates.

Definition 2 *Two predicates $p1$ and $p2$ are mutually exclusive if they are never simultaneously true, i.e., $p1 \wedge p2 = false$.*

To find if two given predicates $p1$ and $p2$ are mutually exclusive, the PHG is traversed backward along all paths from $p1$ and $p2$. Then, a set of *merge nodes* is obtained by picking the node where two backward traversals first meet. $p1$ and $p2$ are mutually exclusive if two backward traversals from $p1$ and $p2$ merge from complementary edges at all merge nodes.

Definition 3 *A predicate p is said to be* covered *by a set of predicates G if $p = true \Rightarrow \exists p' \in G$ such that $p' = true$.*

For a given instruction I associated with a predicate p, its predicate-covering predecessor instructions are obtained by scanning backward the given instruction sequence. An instruction I' associated with a predicate p' is a predicate-covering predecessor of I if p and p' are not mutually exclusive and p' is not already marked as covered in the PHG. After placing I' into a predicate-covering predecessor set of I, p' is marked as *covered* in the PHG and the newly covered predicate is propagated in the PHG to mark other covered predicates.

Definition 4 *A definition d guarded by a predicate p* reaches *a later use u guarded by a predicate p' in the same basic block if p and p' are not mutually exclusive and p' is not covered by any subset of predicates guarding instructions between d and u.*

In other words, d reaches u if d is a predicate-covering predecessor of u.

3.2 Eliminating superword predicates

In this section, we show how to remove superword predicates while preserving the semantics of the original program through the use of *select* operations. Figure 4(a) shows an example sequential code. After if-conversion and parallelization, the control flow is removed and some instructions are guarded by superword predicates shown in parentheses as in Figure 4(b). The first instruction defines a superword predicate Vp and its complement Vnp. A field of Vp is set to true if the result of the comparison is true, and the fields of Vnp are set to the complement of the corresponding fields of Vp. To generate the final code, it is incorrect to simply remove the superword predicates; for example, the first definition of Va would be killed by the second definition. Instead, we rename the second definition and use a *select* instruction to merge their values into one superword variable as shown in Figure 4(c).

Algorithm SEL: Given a sequence of predicated instructions IN, remove superword predicates from all superword instructions by generating *select* instructions.

```
Build a predicate hierarchy graph(PHG)
DU-chain and UD-chain are built based on Definition 4 using IN and PHG
for each definition $d : V = s_1$ op $s_2$ $(P)$
    NeedSelect ← false
    for each use $u \in$ DU-chain(d)
        if ( $\exists$ definition $d_1 \in$ UD-chain(u) such that $d_1$ precedes $d$ in basic block )
            NeedSelect ← true
            remove the predicate of $d_1$
    if ( NeedSelect == true )
        rename $V$ to $r$ in $d$ so that $d : r = s_1$ op $s_2$
        remove the predicate $P$ of $d$
        Insert “$d_{new}$: V = select($V$, $r$, P)” after $d$
        Replace $d$ and $d_1$ with $d_{new}$ in UD-chain and DU-chain.
```

Figure 5. An algorithm to generate *select* instructions

Figure 5 presents the algorithm that generates the minimum number of select instructions required to preserve the original program's behavior. A select instruction is required for some but not all definitions of superword variables, as will be discussed below.

Given a parallelized code with instructions guarded by predicates, we first build a predicate hierarchy graph (PHG) as defined in Section 3.1 [20]. At this stage, instructions guarded by both scalar predicates and superword predicates can be intermixed. For clarity, the reader can assume that the PHG discussed in this section contains only superword predicates. Our implementation actually has separate PHGs for superword and scalar predicates, with connections between the two graphs.

The algorithm relies on both the PHG and use-definition (UD) chains [1], extended in Definition 4 to consider the effects of predication. Using the PHG and the notion of reaching definition (Definition 4), we build DU-chains for the superword definitions and UD-chains for the corresponding uses as shown in Algorithm `SEL` of Figure 5. Although the PHG involves both scalar and superword predicate variables, only superword variables are included in the DU-chains and UD-chains. To correctly handle *upward exposed uses*, all variables are assumed to be defined on entry of the basic block, and these definitions are included when appropriate in the DU-chains and UD-chains. In this way, the compiler can generate a select instruction when there is an upward exposed use.

The main loop of the algorithm `SEL` examines each instruction in textual order. An instruction with definition d needs a select instruction if d reaches at least one use u that is also reached by an earlier definition d_1. If a definition d is the only definition reaching all its reachable uses, it needs not be combined with anything. Figure 4(c) illustrates this point. The first select instruction is not necessary.

Excluding store instructions, this algorithm generates the minimal number of select instructions. Given n definitions to be combined, this algorithm generates $n - 1$ select instructions. The minimality can be proven by reducing the definitions to leaf nodes of a full binary tree.

3.3 Unpredicate

After superword predicates are removed and replaced with select instructions, the code may still contain predicated scalar operations. The simplest way of removing scalar predicates is to convert each predicated instruction into an `if`-statement containing one statement, as in the example code in Figure 6(b). While correct, the code contains numerous redundant conditional branches, six in this case.

Figure 7 presents our algorithm that generates the control flow graph(CFG) representing the improved code as shown in Figure 6(c), given input instruction sequence IN. The main algorithm, called UNP, is shown in Figure 7(a). In addition to deriving the final control flow graph, UNP derives as an intermediate result a reordered instruction sequence IN.

UNP starts by building a predicate hierarchy graph, PHG. Note that this is not the same PHG from the previous section, which contained a mix of superword and scalar predicates. The superword predicates have been eliminated and replaced with *select* operations. Only scalar predicates remain in the new predicate hierarchy graph. UNP also constructs a data dependence graph for instruction sequence IN, capturing the ordering constraints on the instruction sequence.

Subsequently, UNP initializes the CFG with a root node associated with a null predicate P0. The main loop iterates through the input instruction sequence IN. First, we find a set of existing basic blocks where it is safe to insert the

```
bred[i] = fred;         (p)
bred[i] = 100;          (¬p)
bgreen[i] = fgreen;     (p)
bgreen[i] = 100;        (¬p)
bblue[i] = fblue;       (p)
bblue[i] = 100;         (¬p)
```

(a) Predicated scalar code

```
if(p == 1) bred[i] = fred;
if(p == 0) bred[i] = 100;
if(p == 1) bgreen[i] = fgreen;
if(p == 0) bgreen[i] = 100;
if(p == 1) bblue[i] = fblue;
if(p == 0) bblue[i] = 100;
```

(b) Naive unpredicate applied

```
if(p){
    bred[i] = fred;
    bgreen[i] = fgreen;
    bblue[i] = fblue;
}else{
    bred[i] = 100;
    bgreen[i] = 100;
    bblue[i] = 100;
}
```

(c) Improved

Figure 6. Restoring control flow

given instruction. An instruction I guarded by predicate P can be inserted in basic block B associated with predicate P' if $P = P'$ and there is no data dependence preventing insertion of I into B. If the set is not empty, the instruction I is inserted at the end of the earliest such basic block B. Also, in the input instruction sequence IN, I is moved next to instruction I' that is the immediate prior instruction in B. Although we have already processed I, moving it in the instruction sequence will facilitate finding predicate covering basic blocks in Algorithm PCB for subsequent instructions in the stream. If the instruction cannot be inserted into any existing basic block, we create a new basic block B' and I is placed into B'.

When a new basic block B' is created by Algorithm NBB, the predicate covering basic block algorithm (PCB) is used to find a set of predecessors of B'. Whereas Mahlke's *predicate CFG generator* scans forward to find a set of successors, we scan the input instruction sequence backward to find predecessor instructions whose predicates cover the predicate of the given instruction. Since the instructions in the input instruction sequence are processed sequentially, all predecessor instructions chosen must have been inserted already. By keeping a pointer from the inserted instructions to the basic blocks, the predecessor basic blocks for the new basic block are identified. We create a copy *PHG*$'$ of predicate hierarchy graph *PHG* so that we may mark covering predicates during the search for the appropriate basic blocks to connect to the new basic block in the intermediate CFG. The function `does_cover(P', P, PHG')` checks if P' covers P in *PHG*$'$. If P' is not marked yet in *PHG*$'$ and P' is not mutually exclusive with P, the function returns *true*. The function `mark(PHG', P')` places a mark on a predicate node P' in *PHG*$'$ as covered and checks if the predecessor nodes and the successor nodes of P' are also covered as a result of marking. If a node is newly marked, this process is recursively applied to the neighbors of the node. The function `is_covered(PHG', P)` examines *PHG*$'$ and returns *true* if P is marked as covered.

4 Parallelization and Code Generation

In this section, we discuss additional implementation extensions to SLP beyond the two algorithms in Section 3 required to obtain the results in the next section.

Type conversions. A common feature of multimedia applications is type size conversion, particularly to promote small data types before or after arithmetic operations. In the original SLP compiler, if an operation is to be performed on two superwords whose element's type sizes are different, the elements of the superword with the smaller type size is unpacked into a set of scalar variables, the type conversion is applied to the scalar variables, and the converted elements are packed into a set of superwords. Type size conversion is more difficult on superwords than scalar data types, due to alignment issues, instruction set limitations and the impact on parallelization. We have extended SLP such that to the extent possible type conversions are performed in parallel. On the AltiVec, the available instructions supporting type conversion convert to fields that are half or double the size. Type size conversions of a factor larger than two must be broken into multiple conversions. The alignment offset of the destination variable is adjusted from that of the source variables. Predicate variables also may require type conversions so that they match the size of the destination variable of the instruction being guarded. In our benchmarks in the next section, `MPEG2-dist1` and `EPIC-unquantize` have type size conversions.

Reductions. A *reduction* operation consists of obtaining a single element by combining the elements of a vector or array, that is, it reduces the vector or array to one element. It appears as a scalar data dependence to the original SLP compiler, and is not parallelized. Of the eight benchmarks used in the experiments presented in section 5, `TM`, `MAX`, `MPEG2-dist1` and `GSM-Calculation` have reduction operations. Our implementation extends SLP to support reductions similar to the standard code generation for reductions in multiprocessors. We create as many private copies of the reduction variable as will fit in a superword. Instead

Algorithm UNP: Given a sequence of predicated instructions IN, introduce control flow into the instruction sequence after removing predicates.

```
PHG ← Build a predicate hierarchy graph
DG ← Build a data dependence graph
CFG ← new basic block(P0)            // root node
for each instruction I ∈ IN in textual order
    B ← {basic block b | ∀ basic block $b' \in CFG$
        ($b'$ is reachable from $b$ in CFG) ⇒
        ($\nexists$ an instruction $I' \in b'$ such that $I$ is dependent on $I'$)}
    if (B == ∅)
        B ← NBB(CFG, PHG, I, IN)
    else
        Move I in IN to next to the last instruction of the
            earliest basic block in B
    Insert I to end of the earliest basic block b ∈ B
return CFG
```

(a) UNPredicate main

Algorithm NBB: Given an instructions I, predicate hierarchy graph PHG, the current control flow graph CFG, and predicated input code IN, generate a new basic block in CFG.

```
P ← predicate of I
b ← new basic block(P)
B ← PCB(P, PHG, CFG, IN, I)
for each $b' \in B$
    generate an edge from $b'$ to b
return b
```

(b) Create a new basic block

Algorithm PCB: Given a predicate P, predicate hierarchy graph PHG, the current control flow graph CFG, predicated input code IN, and an instruction I, return a set of basic blocks that are predecessors of I.

```
RET ← ∅
PHG' ← PHG
I' ← I.previous
while I' $\neq NULL$
    P' ← I'.predicate
    if (does_cover(P', P, PHG') == TRUE)
        RET ← RET ∪ I'.block
        PHG' ← mark(PHG', P')
    if (is_covered(PHG', P) == TRUE)
        return RET
    I' ← I'.previous

RET ← RET ∪ {ROOT}
return RET
```

(c) Predicate covering basic blocks

Figure 7. Unpredicate Algorithm

of assigning each private copy to a coarse grain computation, different private copies are assigned to each consecutive iteration in a round robin fashion so that the private copies are packed into one superword and reduction operations are done in parallel when the loop is unrolled. Outside the parallel loop, the private copies are unpacked and combined into the original reduction variable sequentially.

Unaligned Memory References. The SLP algorithm packs instructions starting from memory references. Two memory references are packed if they are adjacent to each other and they access a constant offset with respect to superword size. Thus, when the start address of a memory reference is unknown or the address is not constant with respect to superword size, it cannot be packed. We loosen these requirements so that two memory references can be packed as long as they are adjacent. As a result, the address of a superword memory reference can be one of *aligned to zero offset*, *aligned to non-zero offset* or *unaligned*. Depending on the kind of alignment, our implementation generates a simple aligned load, a static alignment with two loads, or a dynamic alignment for an unknown alignment. To optimize the cost of packing, we use a technique we have developed called *prepacking* to group desired instructions together, the details of which are beyond the scope of this paper.

5 Experimental Results

This section presents performance data obtained by applying the algorithm presented in Section 3 to eight computational kernels. This section describes the compiler implementation, experimental environment, the kernels used in the experiments and an evaluation of the results.

5.1 Benchmarks and Implementation

Table 1 describes the eight benchmarks used in the experiments, often found in multimedia and applications. Three of the codes, `MPEG2-dist1`, `EPIC-unquantize` and `GSM-Calculation` are individual functions from three codes in the UCLA MediaBench. `MPEG2-dist1`, `EPIC-unquantize` and `GSM-Calculation` take 55 %, 25 % and 49 % of the run time in `MPEG2` encoder, `EPIC` decoder and `GSM` encoder, respectively. Since this paper focuses on parallelizing loops in the presence of control flow, each benchmark contains at least one conditional.

5.2 Methodology

The Stanford SUIF compiler is the underlying infrastructure [11] for our implementation. We have implemented the

Name	Description	Data Width	Input Size
Chroma	Chroma keying of two images	8-bit character	Large: 400 × 431 color image(1 MB) Small: 48 × 48 color image(12 KB)
Sobel	Sobel edge detection	16-bit integer	Large: 1024 × 768 gray scale image(3 MB) Small: 1024 × 4 gray scale image(16 KB)
TM	Template matching	32-bit integer	Large: 64 × 64 image, 72 32 × 32 templates(1.4 MB) Small: 16 × 64 image, 1 16 × 32 templates(10 KB)
Max	Max value search	32-bit float	Large: 2 100 × 256 × 256(52 MB) Small: 2 8 × 256 (16 KB)
transitive	Shortest path search	32-bit integer	Large: 2 1024 × 1024 (8 MB) Small: 2 16 × 16 (2 KB)
MPEG2-dist1	MPEG2 encoder (dist1 function)	8-bit character 32-bit integer	Large: data blocks for the first 1000 calls (11 MB) Small: data blocks for the first 2 calls(22 KB)
EPIC-unquantize	EPIC(Efficient Pyramid Image Coder) (unquantize_image of unepic)	16-bit integer 32-bit integer	Large: reference input (393 KB) Small: first 4 calls (6 KB)
GSM-Calculation	GSM(Global System for Mobile Communication) (Calculation_of_the_LTP_parameters of gsmencode)	16-bit integer 32-bit integer	Large: reference input (1.1 MB) Small: first 50 calls (16 KB)

Table 1. Benchmark programs

components inside the dotted line of Figure 1 and integrated them in SUIF. The output from the SUIF portion of the system is an optimized C program, augmented with special superword data types and operations. The resulting code is compiled by a GCC (version 2.95) backend which has been modified to support superword data types and operations for the PowerPC Altivec. The resulting optimized kernels are executed on a 533 MHz Macintosh PowerPC G4, which has a superword register file with 32 128-bit registers, a 32 KByte L1 cache and an 1 MByte L2 cache.

Figure 8 illustrates the experimental flow. The `Baseline` version of each kernel is its corresponding initial sequential code. The `SLP` version is derived using MIT's SLP compiler. Version `SLP-CF` represents the final optimized code generated using SLP plus the new optimization passes corresponding to our algorithm. All programs were compiled by the extended GCC backend with optimization flag `-O3`.

5.3 Performance Measurements

Figures 9(a) and 9(b) show speedups of the eight kernels with respect to their baseline versions for two different data set sizes. In Figure 9(a), we use the standard data input size for each kernel, whose footprint is much larger than the L1 cache size. A smaller data set size that fits in L1 cache is used for the results in Figure 9(b), to help isolate the potential gains from `SLP-CF` (or `SLP`) from the effects of the memory behavior of the kernels. For reference, we also show the results of `SLP` for both data set sizes.

For the large data set sizes of Figure 9(a), the speedups achieved by `SLP-CF` range from 1.10X to 2.62X, with an average of 1.65X. All kernels show significantly increased speedups for the smaller data input sizes, ranging from 1.97X to 15.07X, with an average of 5.19X. These results suggest that applying locality optimizations and `SLP-CF` together may result in much better performance for large data sets.

The `SLP-CF` versions of `Chroma`, `Sobel`, and `EPIC-unquantize` all effectively exploit the parallelism available in these kernels, yielding speedups of more than 6.21X. In particular, the 15.07 speedup on `Chroma` is because the data type size of the operands is 8 bits, which results in 16 operations on 8-bit objects per superword operation. `TM`, `Max`, `transitive`, `MPEG2-dist1` and `GSM-Calculation` show more modest speedups. `MPEG2-dist1`, `TM` and `GSM-Calculation` have a reduction. In `MPEG2-dist1`, initialization and finalization remain inside a loop body since the reduction variable is used as the test for loop exit. `Sobel` and `TM` also have performance loss due to unaligned memory accesses. We also observe that for the provided input data set size, `TM` has a very low number of true values for the branch parallelized by `SLP-CF`. While in sequential execution the code would branch around the core computation, in `SLP-CF` it must perform the computation on every iteration and merge with prior results using a *select* operation. This additional computation over sequential execution reduces the benefit of the parallelization. The computation in `GSM-Calculation` is not fully parallelized due to a scalar dependence, but a set of statements between the control flow constructs, representing a loop that was manually unrolled, is parallelized by both `SLP` and `SLP-CF`. Even though the code within the control flow construct is not parallelized, the use of predication allowed our compiler to exploit parallelism across what would have been multiple basic blocks, resulting in a bit higher speedup for `SLP-CF`.

Other than `GSM`, we see that the original `SLP` results do not speed up at all over sequential execution, and for `Max` show a significant degradation. The main reason for this is that SLP is unable to exploit *any* parallelism in the

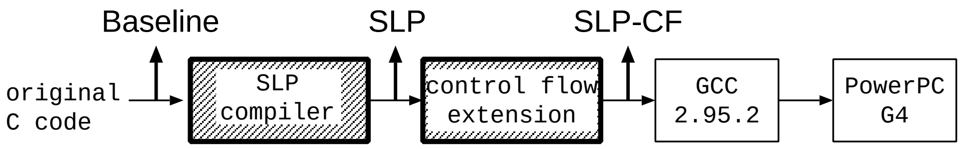


Figure 8. Experimental flow

presence of control flow. In addition, there is some overhead introduced by the SUIF compiler passes leading up to SLP, particularly its code transformations related to dismantling program constructs. This overhead is not inherent to the SLP approach, and we believe it could be eliminated with tuning of the SUIF passes. Nevertheless, since it is not identifying parallelism, the best results we could hope for from the original `SLP` compiler is no change from the sequential performance. The analyses and transformations in `SLP-CF` are crucial to exploiting superword-level parallelism in these codes.

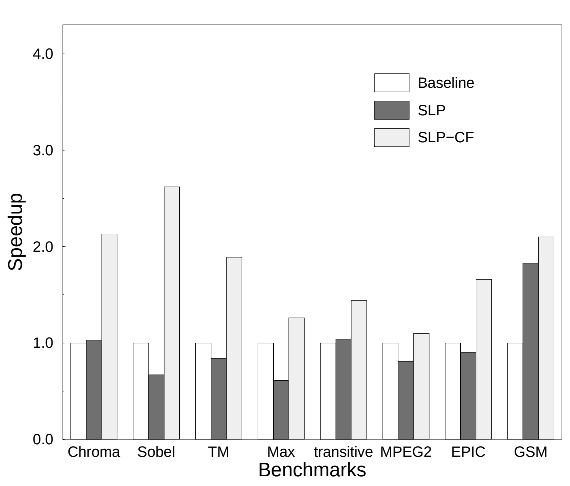
Discussion. The `SLP-CF` approach presented in this paper has demonstrated fairly significant speedups on eight multimedia kernels for which the original `SLP` compiler was unable to exploit much parallelism. In the process of this work, we learned a number of things. The performance gain for superword-level parallelization in the presence of control flow depends on a number of factors, related to both the underlying architecture and the input data set. As was discussed in [24], different instruction set features supporting conditionals impact performance. In the AltiVec, the general mechanism of *select* operations requires executing instructions along all control flow paths and merging the results. When compared to sequential execution, where branches around code constructs may reduce the operation count, it is a tradeoff between parallelism and code with fewer branches versus less overall computation. In examples such as `TM` where the number of branches taken is large, this can limit performance improvement. Also, the AltiVec ISA does not support a full set of general operations for all possible types. As examples, 32-bit integer multiplication, unpacking unsigned integers and division are not directly supported in the ISA, requiring additional instructions. For 16-bit multiplies, `vec_mule` and `vec_mulo` multiply even or odd numbered elements respectively in superword registers, producing two superwords to promote the results to 32 bits. These even and odd multiplications shuffle the data elements breaking the spatial adjacency of data elements, requiring additional instructions to reorganize the results. Bitwise selection causes another problem in conjunction with the inconsistency of scalar boolean value and superword boolean values. Sometimes, the SLP compiler may pack scalar boolean variables into a superword. Since the result of a scalar comparison is either 0 or 1 instead of a vector of all 0s or all 1s, the superword select can be incorrect if scalar boolean variables are packed into a superword and used in selects. Further, locality effects can dwarf the performance benefits of parallelization for memory-bound computations. Since locality optimizations are usually applicable for multimedia codes, optimizations such as prefetching and tiling should be used in conjunction with the parallelization.

6 Related Work

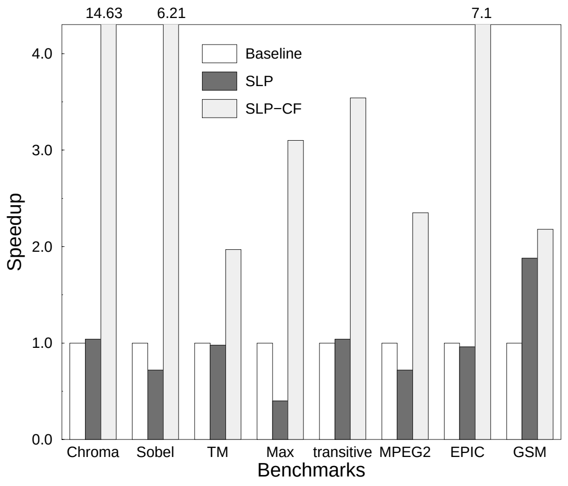
There are several prior works on automatic parallelization for multimedia extensions [16, 15, 25, 5, 19, 4]. Two distinct approaches are used, that is, SLP and an adaptation of vectorization technique. Extending vectorization techniques for conditionals has been addressed [4, 25], but there is no prior work describing how to parallelize conditionals using an SLP-type approach.

If-conversion is described in [3, 2]. More recently, Park and Schlansker describe an if-conversion algorithm that is optimal in terms of the number of predicates used and the number of predicate defining instructions [22]. We use this algorithm in our compiler. Ferrante and Mace describe restoring control flow back from if-converted code [10]. However, their main focus is in generating a sequential code from parallel intermediate representations. Vectorizing compilers targeting multimedia extensions should have a mechanism corresponding to our *unpredicate* unless if-conversion is applied selectively only to the statements that will be parallelized. Mahlke describes a *predicate CFG generator* which restores the original control flow from a predicated hyperblock code [20]. We use his algorithm in the *unpredicate* algorithm when an instruction cannot be inserted into the existing basic blocks.

The *select* instruction is described in many documents [9, 24, 21]. Bik and et. al. use a technique called *bit masking* to combine definitions. However, their method is limited to singly nested conditional statements [4]. Chuang et. al. directly generate *phi-instructions* from the CFG of a scalar code to resolve *multiple-definition problem* in the architectures that support predicated execution [6]. A phi-instruction is a scalar analog of the superword *select* instruction described in Section 2. Their approach is related to ours in that Park and Schlansker's algorithm is also used to derive predicates for the phi-instructions. While



(a) large data set sizes



(b) small data set sizes

Figure 9. Speedups over `Baseline`

phi-predication could be run as a pre-pass to SLP, the code resulting from SLP would potentially contain remaining scalar predicated instructions. In an architecture such as the AltiVec, efficient code generation of the predicated scalar instructions would require an algorithm akin to the unpredicate pass described here. Using phi-predication as opposed to full predication to parallelize conditionals in the SLP compiler is a topic of future research.

7 Conclusion

This paper has described key concepts in extending the notion of superword-level parallelization in the presence of control flow. Because SLP works on basic blocks, we perform if-conversion and derive large basic blocks with predicated instructions to which SLP can be applied. As a result of SLP, the instruction order is modified, and some scalar instructions have been replaced by superword operations, possibly guarded by superword predicates. In many multimedia ISAs, including the PowerPC AltiVec, we must replace definitions guarded by superword predicates with a series of `select` instructions to combine multiple definitions along different paths. We discussed how to minimize the number of `select` instructions that must be inserted. Subsequently, we must restore control flow to the scalar instructions that were predicated by if-conversion and not parallelized by SLP. We described an *unpredicate* algorithm that attempts to restore original control flow whenever possible, to avoid introducing additional branch overhead. We have implemented these algorithms and modified the existing SLP compiler to recognize predicates. The speedups obtained by applying our implementation to eight kernels range from 1.97X to 15.07X.

8 Acknowledgements

The authors would like to thank Samuel Larsen, Mark Stephenson and Saman Amarasinghe at MIT for providing their SLP implementation. This material is based on research sponsored by AFRL and NSA under agreement number FA8750-04-1-0265. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and NSA or the U.S. Government.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Annual Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[4] A. Bik, M. Girkar, P. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[5] G. Cheong and M. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.

[6] W. Chuang, B. Calder, and J. Ferrante. Phi-predication for light-weight if-conversion. pages 179–190, San Francisco, California, 2003.

[7] D. DeVries. A vectorizing suif compiler: Implementation and performance. Master's thesis, University of Toronto, 1997.

[8] J. Draper, J. Chame, M. Hall, C. Steel, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. Kang, I. Kim, and G. Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 26–37, June 2002.

[9] J. Draper, J. Sondeen, and C. Kang. Implementation of a 256-bit wideword processor for the data-intensive architecture (diva) processing-in-memory (pim) chip. In *28th European Solid-State Circuits Conference*, Florence, Italy, September 2002.

[10] J. Ferrante and M. Mace. On linearizing parallel code. In *Annual Symposium on Principles of Programming Languages*, pages 179–190, 1985.

[11] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[12] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.

[13] Intel. *Intel(R) Itanium Architecture Software Developer's Manual*, October 2002. 24531904.pdf.

[14] Intel. *Intel(R) Itanium(R)2 Processor Reference Manual*, April 2003. 25111002.pdf.

[15] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[16] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[17] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[18] C. Lee, M. Potkonjak1, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[19] R. Lee. Subword parallelism with max2. *ACM/IEEE international symposium on Microarchitecture*, 16(4):51–59, August 1996.

[20] S. Mahlke. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana IL, September 1996.

[21] Motorola. *AltiVec Technology Programming Environments Manual, Rev. 0.1*, November 1998.

[22] J. Park and M. Schlansker. On predicated execution, May 1991. Software and Systems Laboratory, HPL-91-58.

[23] J. Shin, J. Chame, and M. Hall. Compiler-controlled caching in superword register files for multimedia extension. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[24] J. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *International Symposium on Computer Architecture*, 2000.

[25] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.